Neo: A Learned Query Optimizer

Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, Nesime Tatbul

Presented by David Adeboye

Why is building a query optimiser hard?

Building a good optimiser today takes thousands of person-engineering hours, and need to be tediously maintained especially as the system's execution and storage engines evolve

As a result, none of the freely available open-source query optimisers come close to the performance of commercial optimisers offered by IBM, Oracle or Microsoft

---

Introducing Neo.

The world's first entire learned query optimiser.

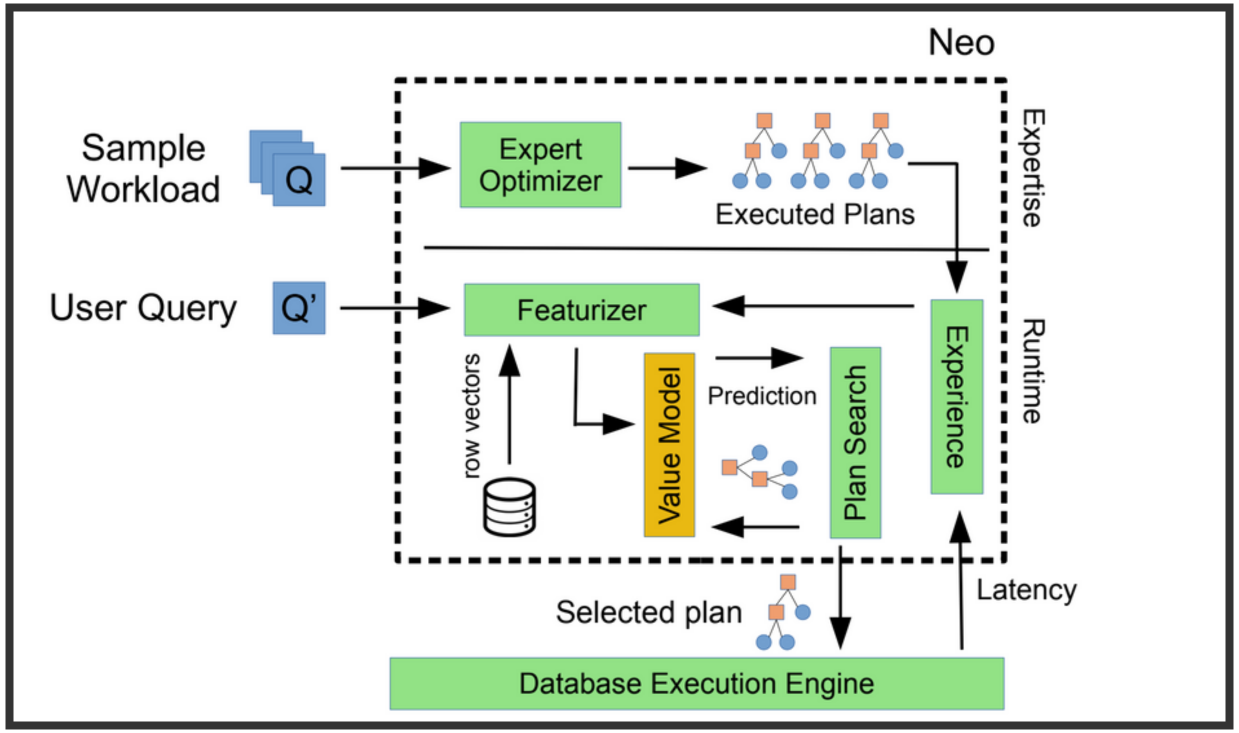
Saikia et al. **Comparative Performance Analysis of MySQL and SQL Server...**

Neo

World's first entire learned query optimiser

Achieves similar or better performance compared to state-of-art commercial optimisers (Oracle and Microsoft) on their own query execution engines

Neo uses reinforcement learning to predict the latency of different query plans.



Neo

Expertise Collection

Generates query execution plans from a traditional query optimizer

Model Building

Neo builds an initial Value Model, a DNN designed to predict the final execution time of a plan

Plan Search

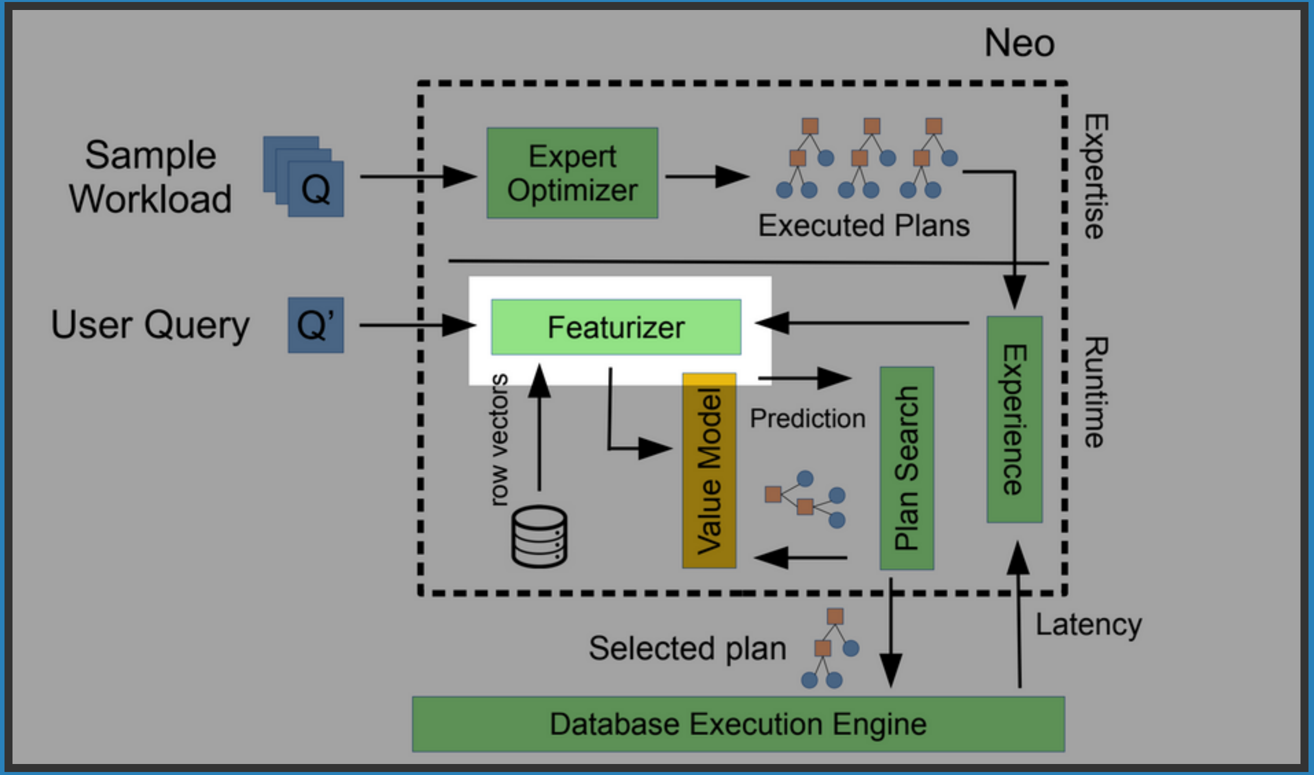
Neo uses the value model to search over the space of query execution plans.

Model Refinement

Using new queries, the model is improved tailored to the underlying database and execution engine.

Query Encodings

1. Query Encoding
2. Plan Encoding



Query Encoding

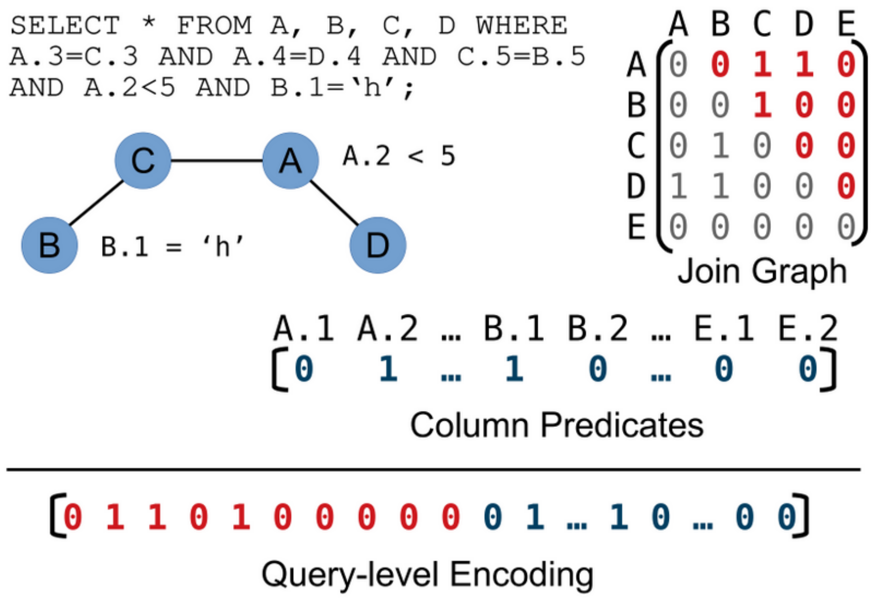
Join Graph

The first component encodes the joins performed by the query, which can be represented as an adjacency matrix of the join graph

Column Predicates

In Neo, three increasingly powerful variants are supported.

- **1-hot**
- **R-Vector**
- **Histogram**



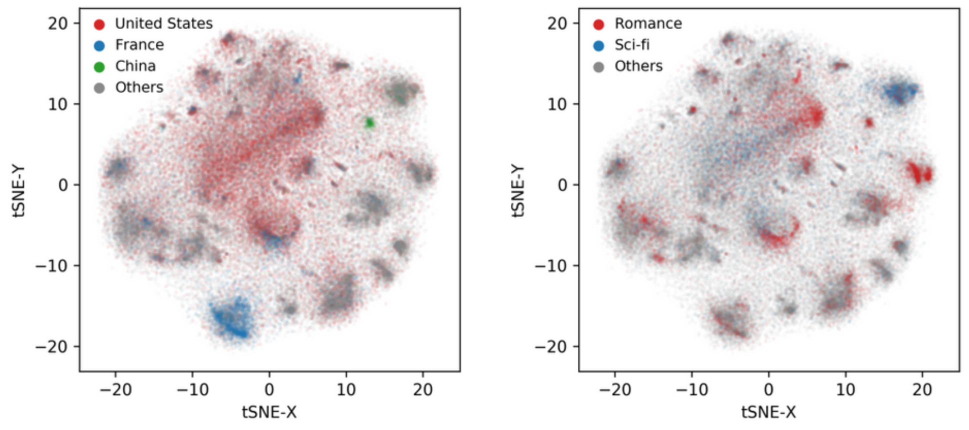
R-vector Featurisation

Purpose: To capture contextual cues among values that appear in a database.

If a keyword **"marvel-comics"** shows up in a query predicate, then we wish to be able to predict what else in the database would be relevant for this query (e.g., **other Marvel movies**).

Row vectors are generated using word2vec, concatenated with number of matched words, number of apperances in training and one-hot encoding of the comparison operators

Variants: Joins & No-Joins



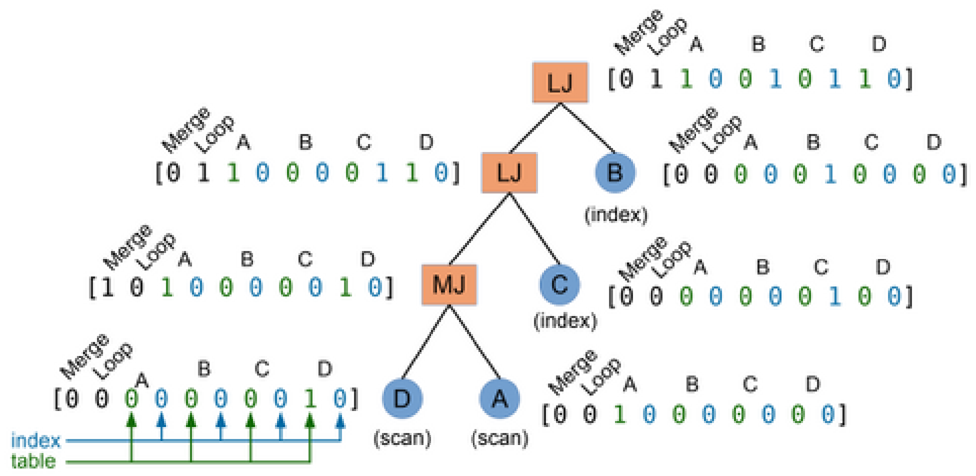
(a) Birthplace of each actor (b) Top actors in each genre

Plan level encoding

Represent the partial or complete query execution plan, preserves the tree structure of execution plans.

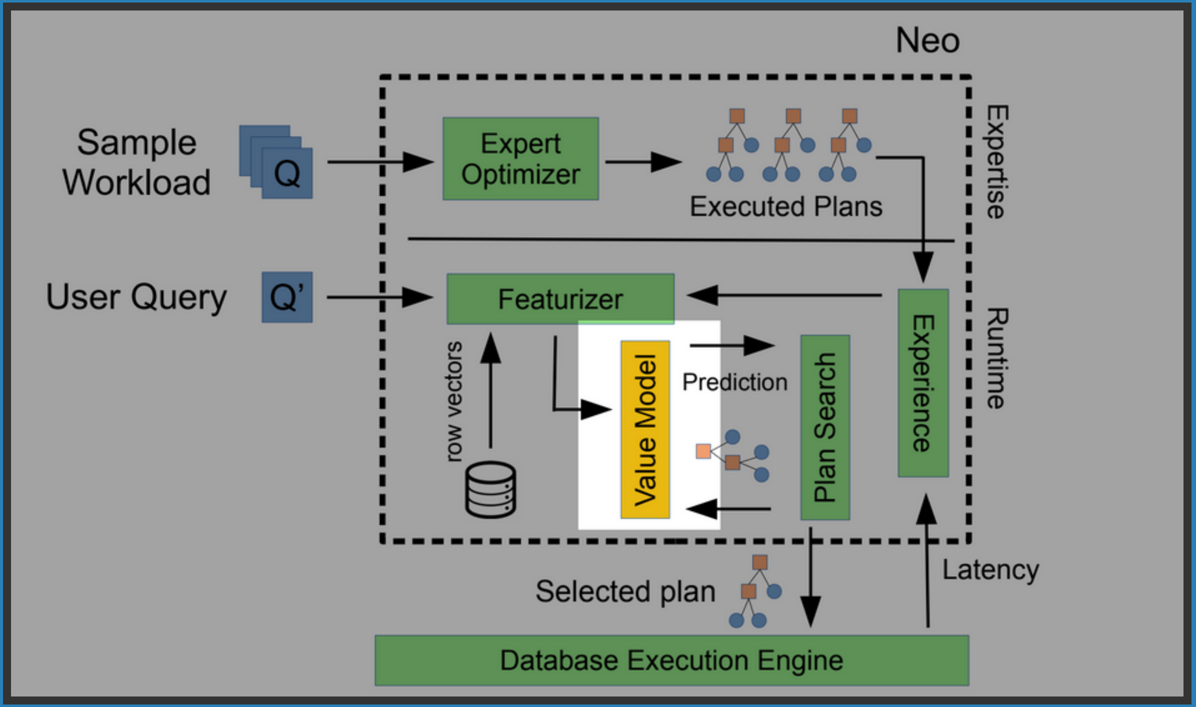
The vectors are created, bottom up, and first store the different join types and then then the type of scan (table, index or unspecified)

Purpose is to provide a representation of execution plans to Neo's value network.

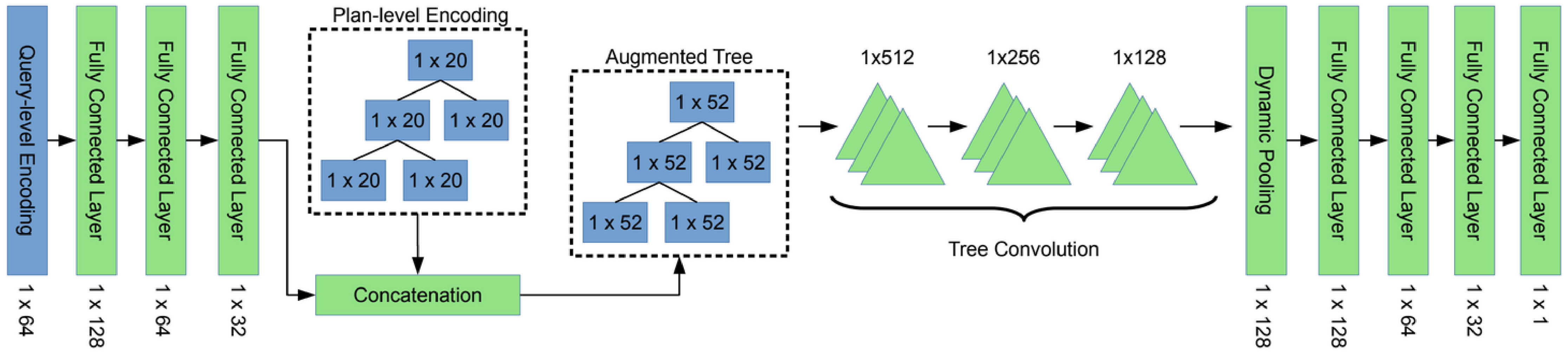


Neo's Value Model

best-possible query latency for an execution plan



Neo's network architecture



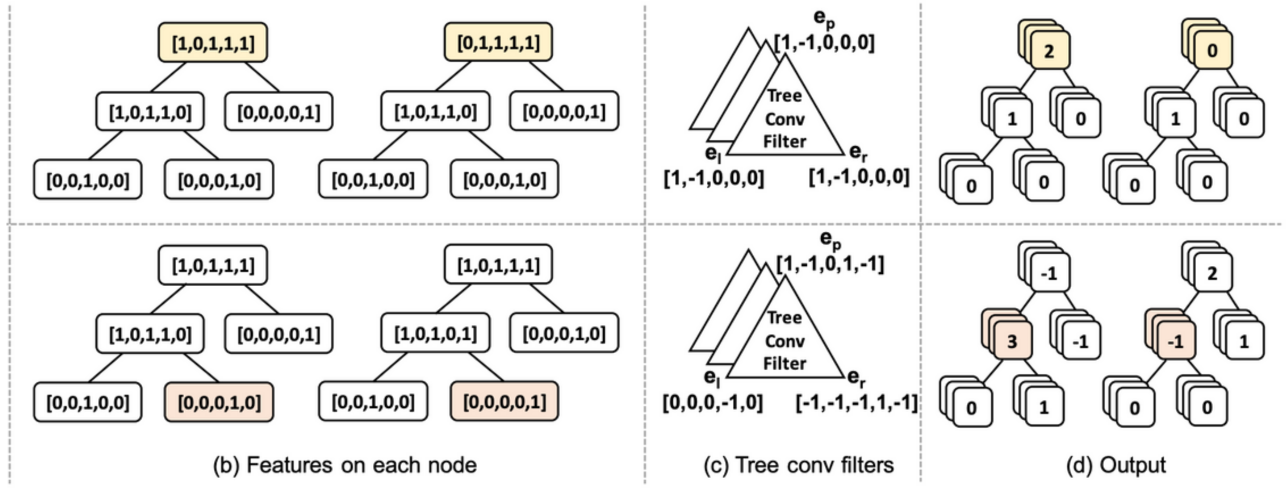
Essentially large bank of optimal patterns that are learned automatically from the data itself, by taking advantage of a technique called tree convolution.

Tree Convolution

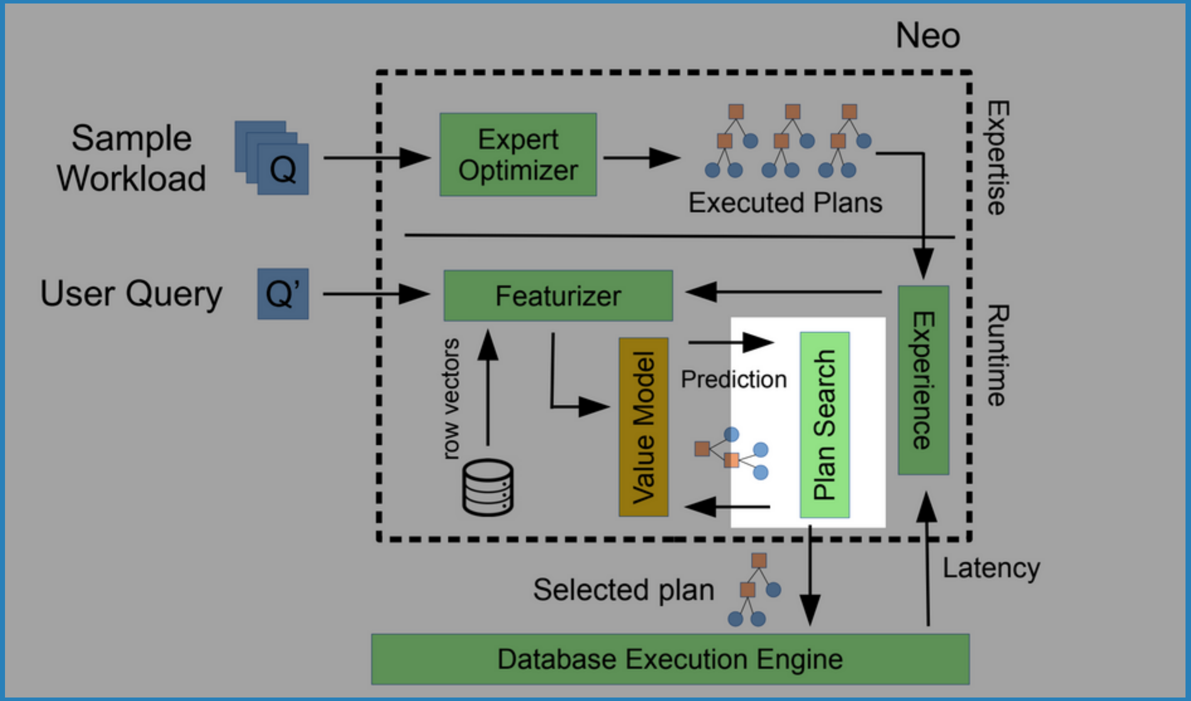
Adaption of traditional image convolution for tree-structured data.

Tree convolution slides a set of shared filters over each part of the query locally, which can capture a wide variety of local parent-children relations.

FIGURE: EXAMPLE OF A TREE FILTER USED TO DETECT TWO MERGE JOINS IN A ROW



Plan Search



Neo
Sample Workload
Q
Expert Optimizer
Executed Plans
Expertise
User Query
Q'
Featurizer
Experience
Runtime
row vectors
Value Model
Prediction
Plan Search
Selected plan
Latency
Database Execution Engine

DNN-Guided Plan Search

Combine the value network with a search technique to generate query execution plans, resulting in a **value iteration technique**

Neo creates a min heap, ordered by the value network's estimation of a partial plan's cost

This heap is initialised with an unspecified scan for each element in each relation for the given query.

Neo iteratively explores the most promising node in the heap.

In the event that the time threshold is reached before a complete execution, Neo explores the most promising child of the last node explored until a leaf is reached

Users can also opt for a 'anytime search' where the algorithms tries to find better results until a fixed time cutoff

Experiments Setup

Three database benchmarks

JOB: Join order benchmark, with a set of queries over the IMDB dataset consisting of complex predicates, designed to test query optimisers

TPC-H: standard database benchmark, suite of business oriented ad-hoc queries and concurrent data modifications. Built to have broad industry-wide relevance.

Corp: 2TB dataset together with 8000 unique queries

Database engines: PostgreSQL, SQLite Microsoft SQL Server for Linux, Oracle 12c

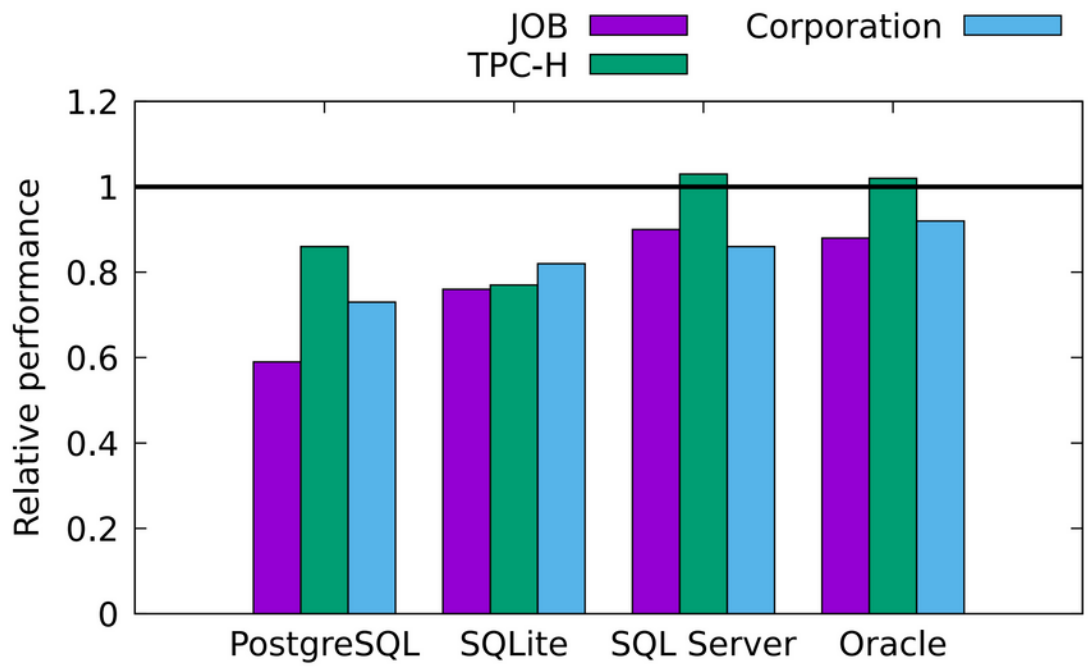
Always used the PostgreSQL optimizer as the 'expert'

100 iterations of training

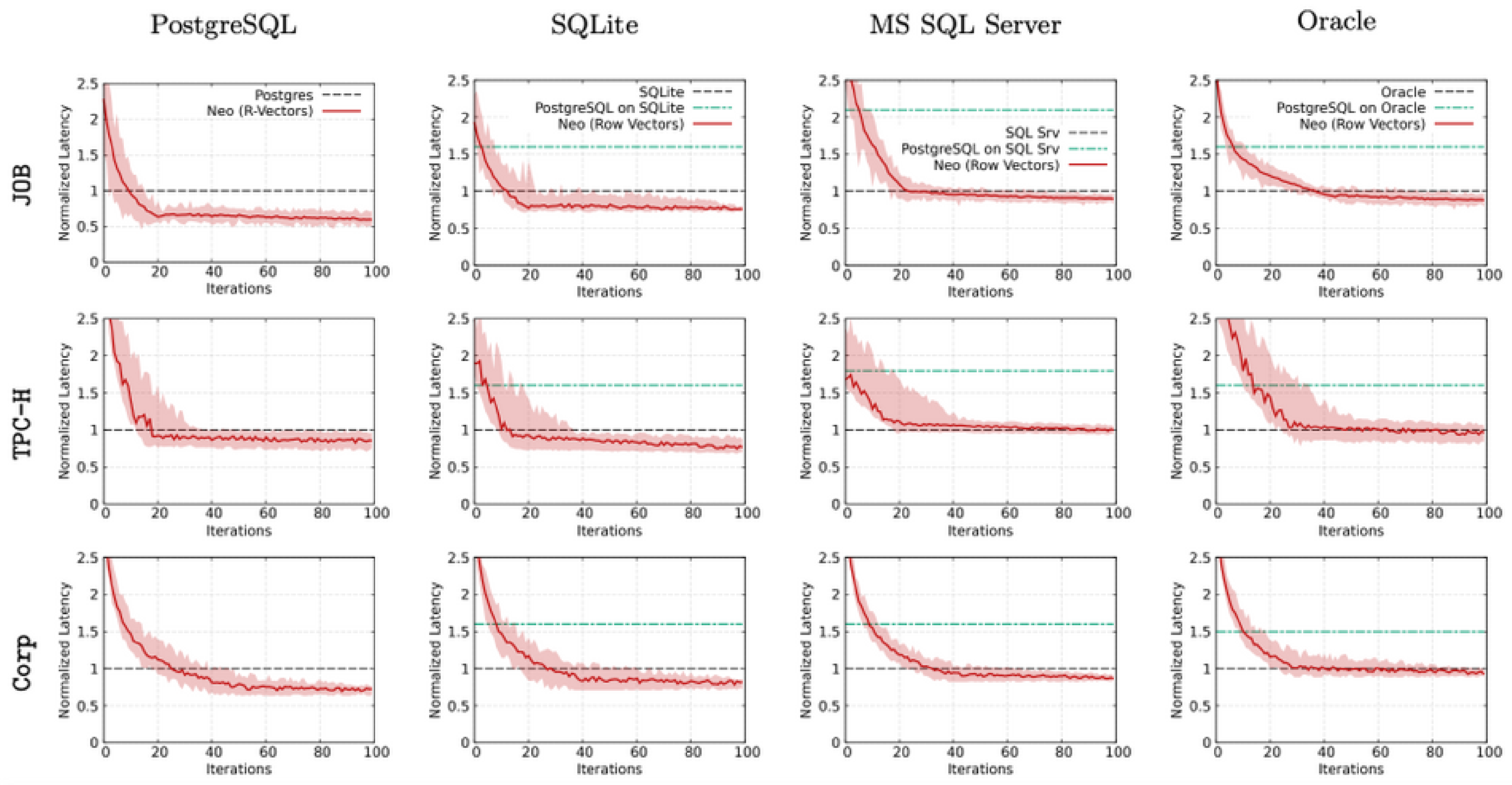
Results: Overall Performance

Neo is able to create plans comparable to both open-source and commercial databases

FIGURE: MEDIAN RELATIVE QUERY PERFORMANCE TO PLANS CREATED BY THE NATIVE OPTIMIZER (LOWER IS BETTER)



Results: Training Time



Results: Training Time

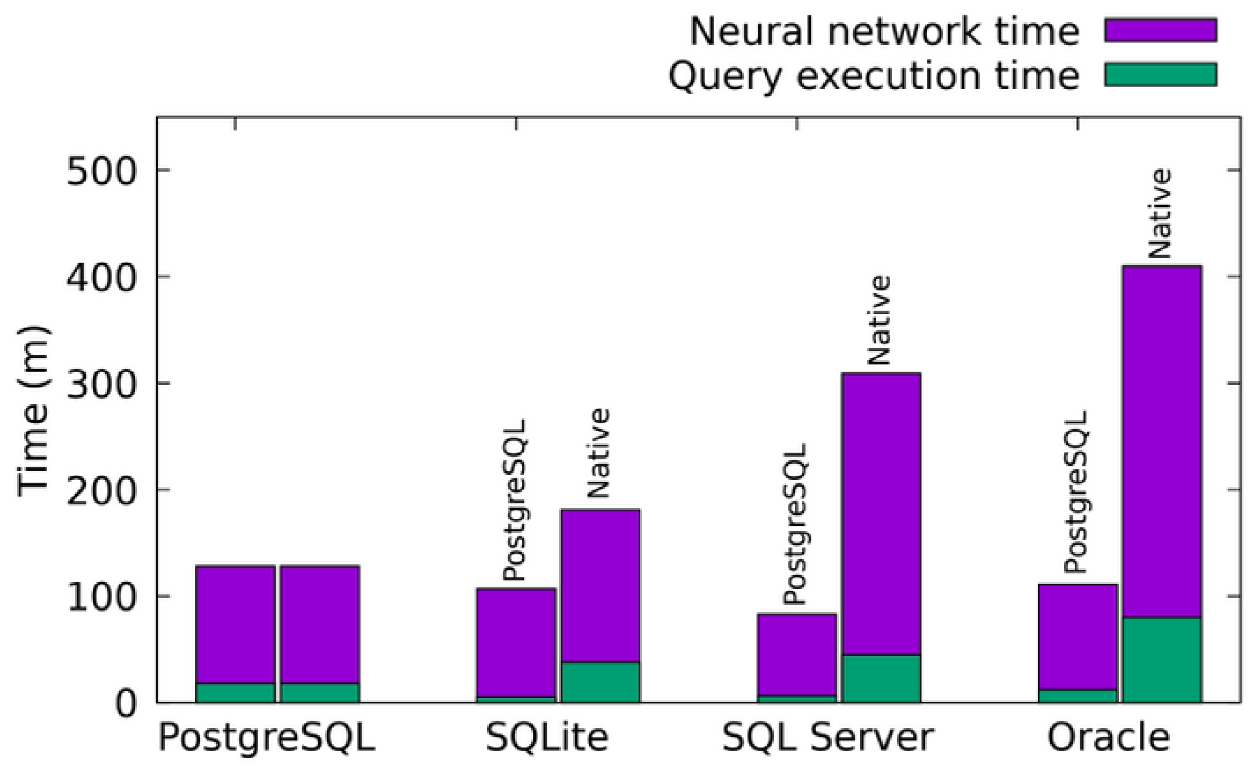
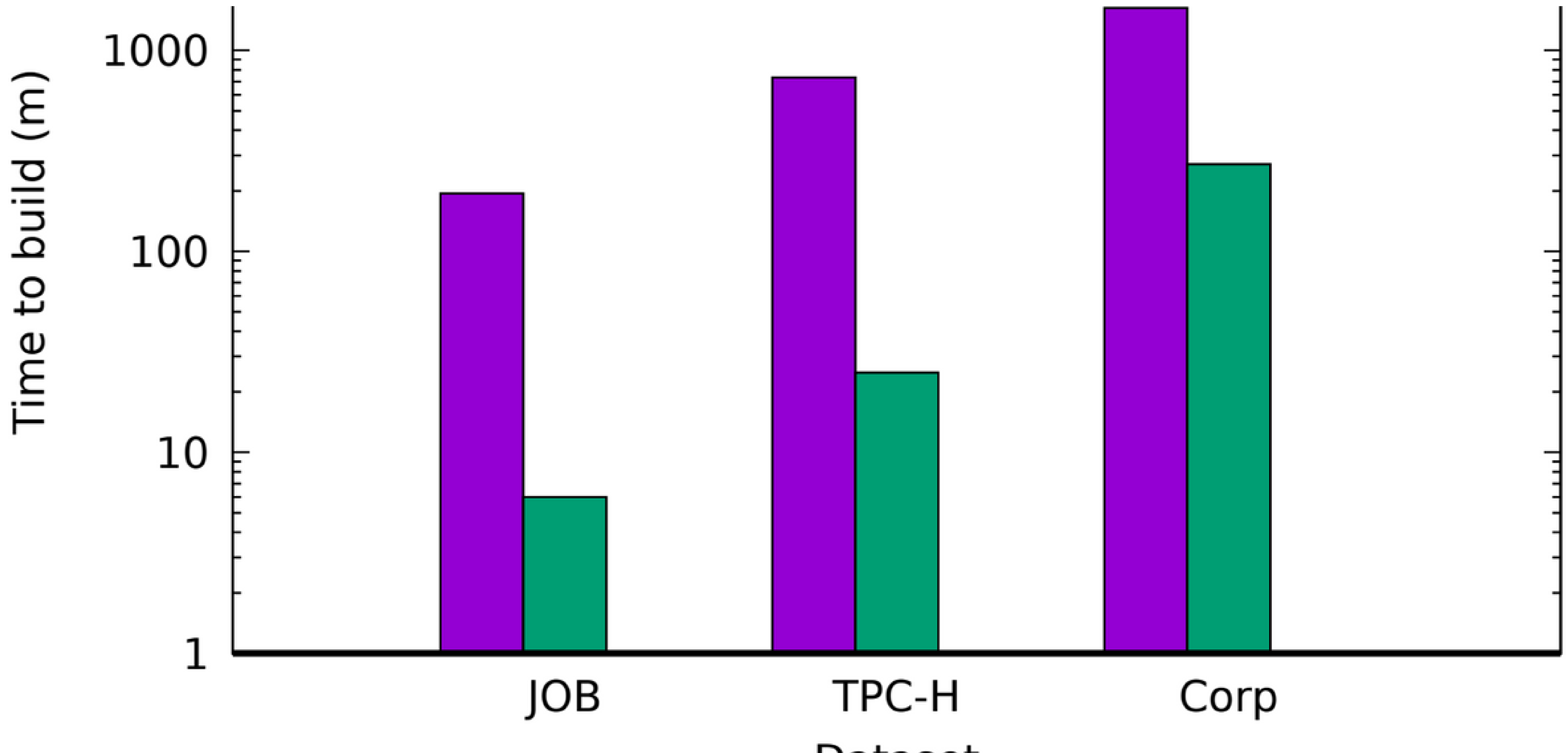


Figure 11: Training time, in minutes, for Neo to match the performance of PostgreSQL and each native optimizer.

Results: Training Time



Limitations

1. **A-priori knowledge**

   Requires knowledge about all possible query rewrite rules

2. **Engine-specific**

   Optimizer does not yet generalize from one database to another, as the features are specific to a set of tables

3. **Query restrictions**

   Neo is restricted to project-select-equijoin-aggregate-queries

4. **Requires a previous optimiser**

   Requires a traditional (weaker) query optimiser to bootstrap its learning process.

Positives

- Really cool idea and it works™
- Extensive evaluation
- Avoid sample inefficiency
- Cost function flexibility

M. Schaarschmidt et al. **Reinforcement Learning in Computer Systems by Learning From Demonstrations.**

Criticism

- **Training phase**
  Loses plug-and-play ability, long training times, R-vector training time
- **Systems evaluation of the system**
- **Ambiguity of Experience**
- **Static cut-off point**
- **Error bars**
- **R-vectors aren't changed after training**
- **R-Vector assumptions**
- **They keep referencing a paper which I couldn't find**

Related Work

Stillger et al. LEO - DB2's LEarning Optimizer (2001)

Leo learns from its mistakes by adjusting its cardinality estimations over time. It requires a human-engineered cost model, a handpicked search strategy, and developer-tuned heuristics.

Marcus et al. Deep Reinforcement Learning for Join Order Enumeration (2018)

Use reinforcement learning combined with a human-engineered cost model to automatically learn search strategies to explore the space of possible join orderings.

Kraska et al. The Case for Learned Index Structures (2018)

Using neural nets we are able to outperform cache-optimized B-Trees by up to 70% in speed while saving an order-of-magnitude in memory over several real-world data sets.

Thanks for listening

How to optimise queries: Joins

SQL Server has 3 types of joins

1 Nested loops joins

If one join input is small (fewer than 10 rows) and the other join input is fairly large and indexed on its join columns.

2 Hash joins

Hash joins can efficiently process large, unsorted, nonindexed inputs.

3 Merge joins

If the two join inputs are not small but are sorted on their join column (for example, if they were obtained by scanning sorted indexes), a merge join is the fastest join operation.

R-vector cardinality example

Table shows the similarity between the vectors for keywords and genres and their true cardinalities in the dataset.

PostgreSQL, with its uniformity and independence assumptions always estimates the cardinalities for the final joined result to be close to 1.

For the example query, Neo decided to use hash joins instead of nested loop joins, and as a result was able to execute this query 60% faster than PostgreSQL

Keyword	Genre	Similarity	Cardinality
love	romance	0.24	11128
love	action	0.16	2157
love	horror	0.09	1542
fight	action	0.28	12177
fight	romance	0.21	3592
fight	horror	0.05	1104

```
SELECT count(*)
FROM title as t,
     movie_keyword as mk,
     keyword as k,
     info_type as it,
     movie_info as mi
WHERE it.id = 3
AND it.id = mi.info_type_id
AND mi.movie_id = t.id
AND mk.keyword_id = k.id
AND mk.movie_id = t.id
AND k.keyword ILIKE '%love%'
AND mi.info ILIKE '%romance%'
```

Results: Robustness

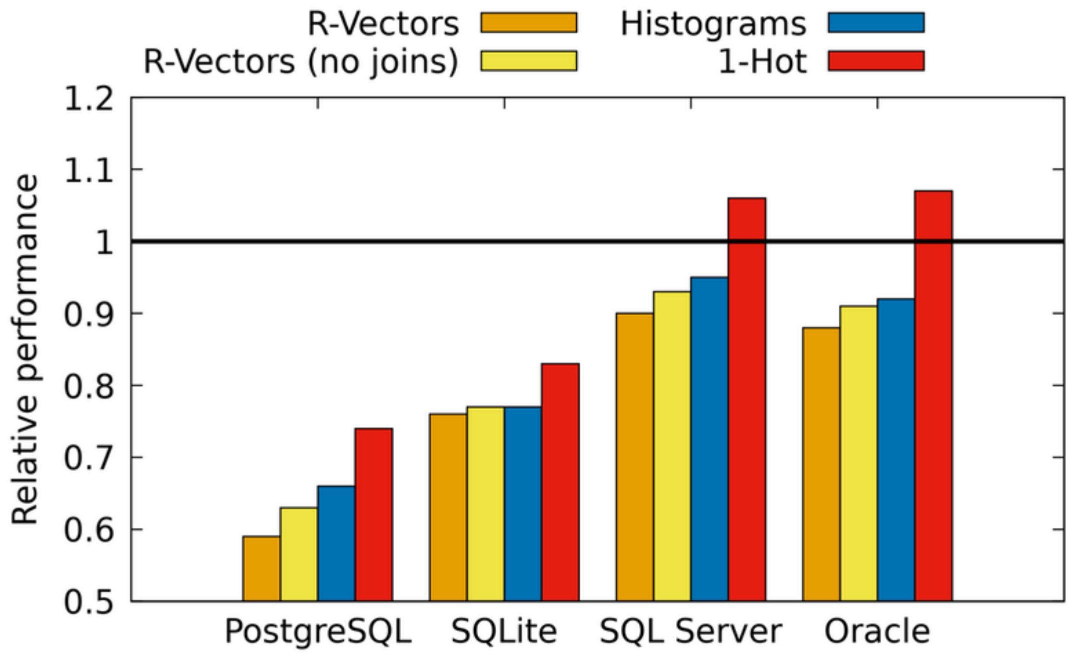


Figure 12: Neo's performance using each featurization.

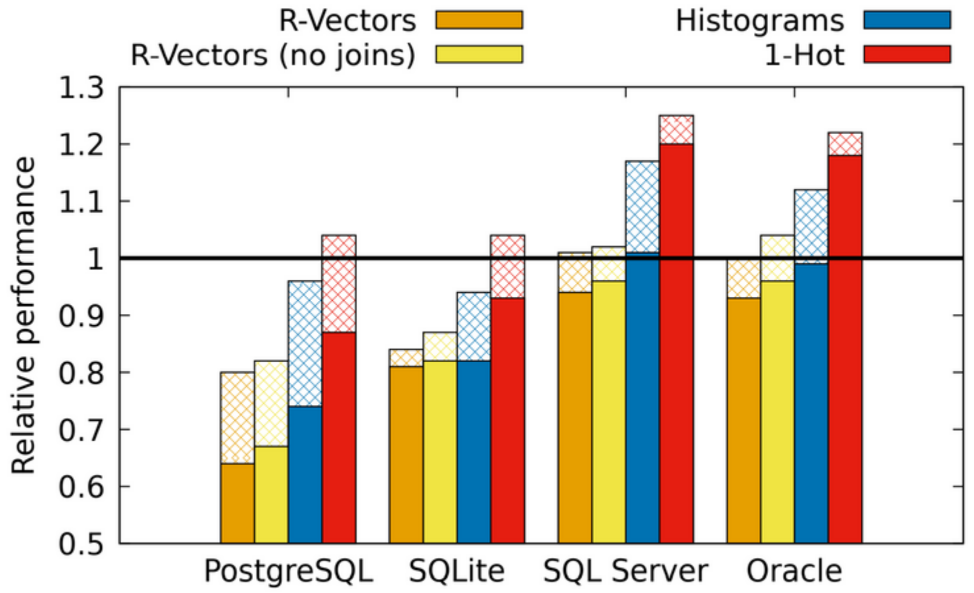


Figure 13: Neo's performance on entirely new queries (**Ext-JOB**), full bar height. Neo's performance after 5 iterations with **Ext-JOB** queries, solid bar height.